NASA Contractor Report 172226

NASA-CR-172226
19840002701

Fault Tolerant Architectures for Integrated Aircraft Electronics Systems

Karl N. Levitt
P. Michael Melliar-Smith
Richard L. Schwartz

SRI International
Menlo Park, California 94025

Contract NAS1-17067

August 1983

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

ENTER:

27 1 1 RN/NASA-CR-172226
DISPLAY 27/2/1
84N10769*# ISSUE 1 PAGE 119 CATEGORY 60 RPT#: NASA-CR-172226 NAS 1.26:172226 CNT#: NAS1-17067 SRI PROJ. 4616 83/08/00 57 PAGES
UNCLASSIFIED DOCUMENT
UTTL: Fault tolerant architectures for integrated aircraft electronics systems
TLSP: Final Report
AUTH: A/LEVITT, K. N.; B/MELLIAR-SMITH, P. M.; C/SCHWARTZ, R. L.
CORP: SRI International Corp., Menlo Park, Calif. AVAIL.NTIS SAP: HC A04/MF A01
Hampton, Va. NASA. Langley Research Center
MAJS: /*ARCHITECTURE (COMPUTERS)/*FAULT TOLERANCE/*FLIGHT CONTROL
MINS: / ADA (PROGRAMMING LANGUAGE)/ AVIONICS/ COMPUTER NETWORKS
ABA: Author
ABS: Work into possible architectures for future flight control computer systems is described. Ada for Fault-Tolerant Systems, the NETS Network Error-Tolerant System architecture, and voting in asynchronous systems are covered.

NASA Contractor Report 172226

Fault Tolerant Architectures for Integrated Aircraft Electronics Systems

Karl N. Levitt
P. Michael Melliar-Smith
Richard L. Schwartz

SRI International
Menlo Park, California 94025

Contract NAS1-17067

August 1983

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Contents

Plates

Chapter 1

Introduction

The goal of the task described in this report is to establish the basis for an advanced fault-tolerant onboard computer that will be the successor to the current generation of fault-tolerant computers (e.g., SIFT [1] and FTMP [2]). Particular features envisioned for this new computer include the following:

- Support for the processing of application programs written in a modern programming language, e.g. Ada
- Minimal burden on the programmer to prepare programs for the fault-tolerant computer
- Flexible, dynamic scheduler
- To the extent possible, an executive that can be easily ported among different processors types
- Immunity to transient faults the number of which might exceed the voting margin
- Immunity to massive transient faults, i.e., that might drive processors to a state from which they cannot proceed without the assistance of other processors
- Extendability significantly beyond that provided by SIFT

- Compatibility with the envisioned electronics system of the aircraft of the future, i.e., a large number of sensors and actuators each with its own microprocessor, and the possibility of replacing a given function that can no longer be processed by one or more backup functions.

Towards this goal we are working on the following technical problems:

1. The use of Ada as the language for the executives of the computer and the application programs – Chapter 2
2. The architecture of a network-based fault-tolerant system – Chapter 3
3. A new paradigm (an extension of the conventional voting paradigm) for comparing the values produced by replicated processors – Chapter 4

On (1), we have identified the potential advantages arising from the use of Ada. Besides the gain in portability, it is likely that the executive can be appreciably simpler than a comparable executive written in other languages (e.g., Pascal or Assembly Code) since the Ada runtime system itself provides some of the basic functions of the executive: scheduling, process synchronization, and memory allocation. The key problems we have been working on are (a) the identification of that data of the application programs that has to be voted on, (b) how to inform the executive when a vote is to take place, and (c) the identification of those points in the program where the amount of voted information is minimized. In the worst case, the amount of data to be voted on can be substantial, including: global variables, local variables, stack frames, multiprocess substructure when a task is composed of interacting subtasks, and the heap that is accessible to these subtasks. A further complication arises when the voted data has to include rendezvous information. Here interactive consistency must be used to ensure that all replicas synchronize with the same subtasks. However, the amount of data is drastically reduced by doing voting when the state of the program does not include values of the global variables, when there are not temporary variables, and when a task has no active subordinate tasks.

On (2), we have identified a preliminary architecture as the basis for the

research. The architecture consists of clusters interconnected by a network. Each cluster, which is logically associated with a sensor, an actuator, or a site of computation, would itself be redundant; the cluster could even be a SIFT computer whose processors are microprocessors. Different from the intracluster interconnection structure, the network that links the clusters would not be star-connected. Instead each cluster could be connected to only a few other clusters (perhaps 3). If each cluster is a SIFT (say composed of 5 processors), then the link between a pair of connected clusters could consist of 5 connections – between corresponding processors in the cluster pair. With this structure, conventional voting could be used to mask errors arising in the transmission of data between directly connected clusters. We have investigated the reliability of such a system, assuming that overall system failure occurs if any cluster exhausts its redundancy or if enough processors fail in any cluster pair such that voted communication between these clusters cannot take place.

We have also studied the problem of finding optimal network graphs. The objective here is to minimize the number of hops required for the transmission of data between any two clusters. This problem appears to be that finding low diameter graphs assuming a constraint on the fan-out for each node.

The assumption underlying (3) is that it might be advantageous to relax the principal concept underlying SIFT (and all voter-based fault-tolerant systems) that all replicas of a task get identical inputs and are expected to produce identical outputs. By relaxing this requirement it would be possible for the replicas to run at different times, thus allowing the system to be less vulnerable to correlated transient faults. If the different replicas of a task produce different values, conventional voting does not work. The voting function is replaced with a *filter* function that, similar to the conventional vote function, takes as inputs the values from the various replicas. We have started to investigate properties for this filter function. It appears that extensions of our clock synchronization algorithm [3] (after elimination of grossly out-of-range values, the median of the remaining values is the clock value to be synchronized to) will work for reasonably well-behaved functions. When the inputs are binary values, the function can be simpler. We do not

yet have a solution when the task function does not satisfy reasonable continuity conditions.

Chapter 2

Application of Ada to Fault-Tolerant Systems

2.1 Scenario and Goals

SIFT successfully demonstrated reliable computation for aircraft flight control applications through replication of flight control programs on multiple computers. Fault masking was achieved by broadcasting results of replicated tasks and majority voting. In SIFT, task replications, executing on distinct processors, maintain loose synchronization using a preplanned schedule.

The characteristics of the SIFT software structure are:

- A fixed set of user tasks
- Tasks are periodic and executed with fixed frequency
- Communication between tasks is limited to a fixed number of "results", broadcast at the end of each task iteration. Tasks share no storage and have no communication during execution.
- Communication between successive iterations of a task is limited to the same broadcast "results". No storage is preserved between task iterations.

- A preplanned schedule ensures that task results are available when required by other tasks; tasks are never required to wait for input values.
- Errors in broadcast results, caused either by processor or communication faults, are masked by majority voting. Since broadcast results are the only information representing task state which is retained, no other form of fault masking is required.
- No form of masking of errors due to incorrect programs is included.
- Errors detected during voting result in reconfiguration.
- Discrimination is provided between solid and transient faults. Transient faults are masked by voting and do not cause reconfiguration.
- Reconfiguration consists of choice of a new, preplanned, schedule and allocation of tasks to processors.
- Based on individual reliability requirements, tasks can be selectively replicated to any necessary degree. These replications can be allocated to processors to balance processor load.

The orientation of the SIFT design is towards predictability and reliability rather than flexibility. An advantage of the approach is the very simple, and therefore inherently more reliable, nature of the Executive software. The simplicity of the approach used in the Executive software imposed many constraints on the user and exposed aspects of scheduling, communication and replication. These constraints are not inherent in the "SIFT concept" – they were imposed to allow a very simple implementation.

In this report, we investigate to what extent the concept of SIFT can support a more general user interface. We consider a system in which the structure of the user program is not constrained by the needs of fault tolerance, and in which the system is not as dependent on the user to specify management of information and resources in the system. In particular, we seek to permit a more dynamic program structure.

To this end, we consider the Ada virtual machine and investigate building a *reliable* Ada machine. Our goals are:

- To provide fault-tolerant support for a wide class of Ada programs.
- Ada programs should be unchanged, except for advisory directives.
- To allow greater asynchrony between executions of Ada program replications.

2.2 Considerations in Providing a Reliable Ada Machine

In investigating a reliable Ada machine, we continue the SIFT approach of replication on independent processors, with error detection and masking based on majority voting. For majority voting to suffice to detect and mask errors, *all replications* of the program executing on working processors are required to produce *exactly the same results.* Even non-deterministic programs must adhere to this requirement – all instances of the program must behave identically.

2.2.1 Effect of Non-determinism

Ada provides a tasking facility, which inevitably introduces non-determinism. This non-determinism results from direct interaction between tasks and from access by tasks to shared global variables. Ada restricts access by tasks to shared global variables, so that all non-determinism in Ada programs stems from direct communication between tasks. The Ada mechanism for interaction between tasks is the *rendezvous.* The rendezvous involves a task which calls a rendezvous *entry* and a task which *accepts* the entry call. An entry call, equivalent to a procedure call (with parameters), suspends the calling task until the entry is accepted and a rendezvous is completed. When the called task accepts the call, a rendezvous occurs, and the body of the accept procedure is executed. Following completion of the accept procedure, both tasks are allowed to continue asynchronously. If a task reaches the accept point, it is suspended until called.

Non-determinism is introduced by several tasks contending asynchronously for the same accept procedure. Ada does not provide any guarantee of timing for the concurrently executing tasks, and thus does not determine which task will reach the call first. This timing can be influenced by lower level factors such as system scheduling, interrupt handling, etc. These factors may vary from processor to processor. In order for our majority vote masking to succeed, we must be able to guarantee that all processors executing the multiprocess Ada algorithm *accept the same entry call* into the rendezvous. We shall refer to this as a *consistent rendezvous.*

Tasks in Ada are *objects*, which may be dynamically created and dynamically terminated. One does not have any predefined configuration of tasks. It is possible for an Ada program to contain an arbitrary number of instances of an Ada task type.

2.2.2 Periodic Voting

The execution of an Ada program, of course, can be of arbitrary duration. Reliability requirements demand periodic voting to detect and mask errors. When should these votes be performed? Each vote in each processor must be performed at exactly the same point in the computation. The moment of voting cannot be determined solely on the basis of time, since different processors may be in different states at that time. It also is not possible to determine for an arbitrary program how to embed vote requests in the program to obtain votes with appropriate periodicity. For an arbitrary program there will be no obvious iterative structure which could guide this choice. One must also take into account that some points in the program may be more appropriate for voting because of cost or effectiveness.

A second issue concerns what information need be voted. Sufficient data must be voted to detect that tasks instances executing on different processors are performing the same computation. In SIFT, because voting is performed on task results at a time when the task has terminated, only those results need be voted.

For a more general Ada program at which votes are taken periodically, there is no explicit indication of what constitutes "results".

2.2.3 Resource Management and Scheduling

Many potential applications of fault-tolerant computing involve real-time performance constraints. In SIFT, these constraints are guaranteed to be satisfied by a rigid, preplanned schedule. An Ada program, designed to meet the same constraints, depends on dynamic interaction between the program and the Ada scheduler. We expect that the Ada programs to be rendered fault tolerant will already contain the resource and scheduling strategies necessary to meet the real-time constraints. The introduction of fault tolerance should not perturb this basic strategy, although there will inevitably be some overhead introduced as a result of the additional mechanism.

2.2.4 Where to Embed Fault-Tolerance Mechanism

The additional mechanism needed to achieve fault-tolerance can potentially be introduced at one of three levels:

- Below the level of the Ada run-time system.
- Within the Ada translator and its run-time support.
- Above the level of the Ada virtual machine.

The first alternative, implementing fault-tolerance below the level of Ada, implies the Ada translator and its run-time support can be completely unchanged. In order to accomplish this, one would have to introduce a fault-tolerant version of the *processor architecture* assumed by the Ada translator. This approach is certainly feasible, using mechanisms such as dual-dual. The problems to be solved turn out to be comparable to those using the other alternatives, but the

mechanisms cannot exploit the structure of the Ada program to reduce the cost of the additional reliability.

The third alternative would consist of an Ada package, programmed entirely in Ada, to implement the necessary mechanisms. This package would reproduce to the Ada user program a fault-tolerant virtual machine equivalent to the original machine. This would allow a highly portable solution, allowing the fault-tolerance mechanisms to be applied to any system supporting the Ada. To accomplish this, all necessary mechanisms would have to be expressible *within Ada*. Thus, the consistent rendezvous must be programmed using the rendezvous facility for communication between processes. Even with the aid of a preprocessor to introduce additional statements into the Ada program, it would still be necessary to augment the Ada compiler and run-time system to provide information not normally accessible to the Ada program.

The second alternative, that of modifying the compiler and run-time support of Ada, permits more efficient implementation of the necessary fault-tolerance mechanisms. This is at the expense of requiring changes to a rather complex compiler and run-time system, and results in a translator-specific implementation of fault-tolerance.

In the following sections, we explore the capabilities necessary to support the fault-tolerance techniques, and comment on the difficulties in implementing fault tolerance within the Ada system.

2.3 Mechanizing Fault-Tolerance for Ada

In this section, we describe mechanisms to support a fault-tolerant Ada virtual machine. It requires both error detection and masking support and a mechanism to ensure consistent rendezvous. In presenting the major ideas, we treat Ada programs, possibly itself implementing a multiprocess algorithm, as a monolithic program, replicated in its entirety.

2.3.1 Error Detection

The basic approach will be based on replication of the Ada program on independent processors. We use a majority voting scheme to detect faults in a minority of processors – the consensus in such a configuration is assumed to be correct.

In SIFT, each processor uses only *voted* values as inputs – achieving immediate error masking. Here, where there is no notion of distinct inputs, we have no concept of masking input values. Rather, we use majority voting to *detect* errors. Each processor uses only local state information in performing its computation. Following an error, a processor will continue to compute erroneously, but cannot influence other processors' computations. The error will be detected during majority voting, leading to later fault diagnosis and reconfiguration. Error masking occurs as a result of reconfiguration to exclude dependence on erroneous processors.

To detect any erroneous computation, it is necessary to vote the *entire* state of the program. Voting any less than the entire state could permit an undetected error that might adversely affect the future computation. The program state, of course, can be rather extensive – consisting of the run-time stack, heap, expression stack, and any other run-time management information. Rather than broadcast and vote this potentially large amount of data, we compute a *signature* of the state. This signature should be an encoding of the state with sufficiently high probability that distinct states map to distinct signatures. Furthermore, signature calculation should not be highly correlated with the computation; if by chance an erroneous chance reduces to the same signature value as that of the consensus, further computation and a further vote should have a low probability that equal signatures will again occur. Digital techniques such as [6] satisfy these criteria. The length of the signature can be adjusted to meet the required probability of immediate error detection.

One consequence of a multiprocess Ada program is that different instances of the same program may contain tasks scheduled differently on different processors.

Programs may *never* be in exactly the same state. Consequently, we cannot in general vote entire Ada programs. Rather, voting must be done on a task-specific basis. Therefore, it must be possible to determine a partitioning of program state into task states. This raises several problems.

First, there may be no simple way to partition global state into task states. Secondly, even when such a partitioning is possible, it may be impractical to deduce. Because processes may interact through global variables, the appropriate partitioning may be only dynamically determinable. In order to provide a practical solution, we will disallow reference to global variables, forcing all task communication to be via rendezvous calls.

For any partitioning, it is necessary that the combined partitions account for the entire global state of the program. Votes at different times on task states must ensure that the net effect is to guarantee that no information will escape being voted. To ensure this, it is necessary not only to vote the information inside the task state, but all information being communicated between tasks. Since all information flow occurs by rendezvous, it is sufficient to additionally vote all values passed as parameters by entry calls. Consistent with our global variable restriction, no **in out** parameters or **access** values may be passed as entry parameters.

Assuming the implementation of Ada is such that each task has a local run-time stack and expression stack, voting the current state of a task necessarily requires voting the value of these stacks. This cannot be done, of course, above the Ada virtual machine, but can be accomplished by functions added to the Ada run-time support. It is not a safe assumption that heap space is partitioned in a similar manner. Any use of the heap, to allocate access variables, for example, must be traceable to a single task. Voting must include all stack and heap values.

Vote values are generated in one of two ways. Upon encountering a user-supplied vote *pragma*, a signature of the task's entire state is computed and broadcast to the other processors. Upon entry call to a rendezvous, signatures of input **in** parameters are computed. Upon return from a rendezvous, signatures for

out parameters are computed. Each signature is tagged with a task identification and a sequence number which uniquely identifies that vote.

Having established that state signatures are broadcast to all processors, we now describe two possible algorithms for detecting and reporting errors. Because no processor is dependent on values computed by other processors, there is no need for synchronization between processors at vote points. However, some synchronization points are necessary in order to avoid an unbounded amount of storage necessary to hold signature values until a consensus is possible.

The simplest algorithm provides storage for one signature per task instance. Processors can proceed asynchronously up to a vote point. No task instance can progress beyond a vote point until all other instances have reached the previous vote point. To ensure that a minority of failing processors cannot indefinitely delay a vote, we must include a timeout mechanism in this vote. Timing starts when a majority of processors have submitted values. It is assumed that it is possible to establish an appropriate timeout value for each task and that the scheduling can maintain the skew between instances of the task on working processors to less than this value. At the expense of increased storage, it is easy to extend this algorithm by storing additional signatures, thereby allowing greater asynchrony.

There is an alternative algorithm that allows much greater asynchrony without storage penalty. Since voting is used for error detection rather than masking, it is not necessary to vote every signature value separately. Rather, we aim to maximize the number of pairwise comparisons between values generated by different processors.

The voter stores, for each pair of task instances, one signature, its tag, and the id of the processor that generated it.

- If no value is currently stored, a signature triple arriving from either processor can be stored.
- If a signature triple arrives from same processor as that of the outstanding triple,that signature is ignored.

- If the arriving signature is from the other processor:
  - it is ignored if its sequence number is smaller than that outstanding;
  - it is stored if its sequence number is greater than that outstanding;
  - it is voted if the sequence number of the arriving triple is equal to the outstanding triple. Error reports are generated and broadcast when discrepancies in the vote are encountered.

To implement this scheme, we require that successive signature values be computed cumulatively, i.e., that the previous signature be included in the calculation of the next. Thus, each vote includes all previously computed signatures, and errors can be detected even though every signature value is not voted independently. As for the simple algorithm, a timeout mechanism must be used to ensure detection of processors that generate no signature or infrequent signatures. Processors that generate signatures with inappropriate sequence numbers will also be detected.

The maximum interval between votes in this scheme is equal to the maximum interval between generation of signatures plus the maximum skew between the execution of the task on different processors.

2.3.2 Error Masking

The voting of task state, as described above, can only be used to detect errors. Reliable operation requires also that error masking be provided:

- to mask transient errors,
- to move task instances from processors deemed faulty to other processors.

The algorithms by which the SIFT Global Executive diagnoses faults from the error reports, and distinguishes solid from transient faults, are equally applicable here and need not be described. In SIFT, voting automatically masks transient errors and the Global Executive need take no action. When the Global Executive

diagnoses a solid fault, tasks must be assigned to execute on other processors. Those processors have already obtained the required input values, with errors masked by prior voting, and can immediately assume the tasks. For the current scheme, however, the Global Executive must issue explicit directives to mask both solid and transient faults. This masking must be performed by copying *the entire program state* from a processor deemed to working correctly. The entire state can be copied at once, though it may be possible to copy on a task by task basis, thus reducing the time for which processing is suspended.

Chapter 3

NETS: Network Error Tolerant System

Introduction

Our purpose in this research is to design and assess a fault-tolerant system that could be the successor to the SIFT and FTMP class of computers. Although SIFT and FTMP provide a reliability that for aircraft computations far surpasses what could be achieved by nonredundant systems and incorporate redundancy in an elegant way so that the reliability can potentially be proven using analytical methods, they are deficient in several ways. They cannot be expanded beyond 8 (or so) processors and are not well-suited to the trend towards using *smart* sensors and actuators distributed throughout the aircraft. SIFT and FTMP were designed to be the *centralized* computer in an aircraft that contains primarily passive and *dumb* sensors and actuators. These computers use distributed computation – distributed over a number of processors – to achieve fault-tolerance. However, the distribution is somewhat degenerate in that a large subset of the processors are performing identical computations.

The computer concept under consideration in this task, NETS (Network Error Tolerant System), is a bona-fide distributed system. NETS is an intercon-

nection of *clusters*, each of which can be a simplex (nonredundant) processor or itself a fault-tolerant computer – say a SIFT configuration of 3-5 processors. NETS offers all of the "conventional" advantages of a distributed system (e.g., expandability, highly-parallel computation, physical separation of computation sites), in addition to advantages particular to the goal of fault-tolerance (e.g., less costly fault-tolerance, some immunity to massive transient faults, and better adaption to fault conditions). We have completed a preliminary design and assessment of NETS.

In the following sections we present:

- The specific goals of an aircraft fault-tolerant system that motivated the design of NETS
- An overview of the NETS architecture
- An assessment of the reliability of NETS
- Design issues to be considered in a follow-on to the current investigation, e.g., the design of the distributed network executive that manages the fault-tolerance for NETS, algorithms to compute optimal communication paths, an approach to handling massive transient failures, and network-wide synchronization requirements.

3.1 Requirements of an Advanced Fault-Tolerant System

The particular requirements for an advanced aircraft computer that motivated the design of NETS are the following:

- **Fault-Tolerance**. We are assuming that, as in SIFT, the probability of a critical computation yielding an incorrect or late result is not to exceed 10^{-10}/hour over a 10 hour period.
- *Believable* **Reliability**. Although we might never formally verify the NETS design or implementation, the approach to fault-tolerance should be easily

understood and verifiable by informal reasoning. Usually, believable reliability is achieved only if there is no single site of computation whose failure could result in system failure, and if the fault-tolerance mechanism are conceptually simple.

- **Very Little Special Purpose Hardware.** Most of the hardware should be commercially available and known to be intrinsically reliable through extensive field use. Special purpose hardware is prone to design and, perhaps, failure in operation when exposed to unexpected environmental conditions. Furthermore, the complexity of most specially developed chips precludes thorough testing on the part of the manufacturer. Most chip designs become reliable only after extensive testing by users followed by modification by the manufacturer. Special purpose designs will not be so thoroughly stressed and, hence, will be much less reliable.

- **Expandable and Contractible.** A family of systems all with the same basic design, but which differ in terms of the number of processors, the size of the processors, and the degree of error coverage is desirable. An order of magnitude or more difference between the smallest and largest system in the family should be feasible.

- **On-line Insertion or Removal of Sites.** It should be possible to change the configuration without disturbing the currently proceeding computations.

- **Immunity to Massive Transients.** No current fault-tolerant system will maintain operation in the presence of faults that impact the operation of more than 1 (or 2) processors – typically the voting margin of most systems. However, power surges or lightning could cause erroneous behavior – albeit temporarily – from a considerable number of processors. A fully distributed system, by virtue of having the processors physically separated by a reasonable distance, should be able to survive transients that impact a large number of processors or temporarily disable a few entire sites.

- **Be Capable of Interfacing to *Smart* Sensors and Actuators Distributed Throughout the Aircraft.** These sensors and actuators might themselves

offer some fault-tolerance and, surely, will provide on-site computation – usually in the form of a microprocessor. An important issue in the design is to organize the computations so that the intersite communication is low compared with the communication within a site.

- **Capability of Using Different Processor Types.** The use of different types is recommended to reduce the probability of correlated faults, which could result in system failure. In addition, the processor should be matched to particular computational needs of the applications.

- **Portable Executive Software.** If a single design for the executive is to be usable for all processor types, it must be portable. Some specialization of the implementation for particular processors can be accepted, but only if the specialized part is relatively small compared with the total software. The key, then, is to design the executive as two components: a *fixed* part and a *variable* part.

- **Be Capable of Handling Critical Real-Time Computations.** The deadlines of critical tasks must be achieved. The interaction of unpredictable tasks compounds the difficulty of demonstrating that task deadlines are satisfied in a distributed system.

- **The Application Programmer Should Not Have to be Concerned with the Fault-Tolerance Mechanisms.** The key is to provide the programmer with an interface that is not dependent on the processor he is producing programs for, on the amount of fault-tolerance required for the computation, and on the location of tasks interacting with his task.

- *Reasonable* **Cost.** In contrast to the situation 10 years ago, the cost of the computer hardware is not necessarily of prime concern, although it is desirable for the system not to be too profligate in its use of hardware. Of more concern, however, is the cost of the software and of the maintenance of hardware and software.

The current SIFT system addresses primarily the Reliability, Believable Reli-

ability, and Real-Time goals. The NETS system concept can satisfy all of the above goals.

3.2 Overview of the NETS Architecture

The high-level organization of NETS, as shown in Figure 1, consists of an interconnection of *clusters.* A cluster is a site which can be associated with a sensor, an actuator, or can be a *computation* cluster whose role is to generate outputs in response to inputs. A *sensor cluster* will have no logical inputs; an *actuator cluster* no logical outputs. A computation cluster will have both inputs and outputs. In generating the value to be delivered to an actuator in response to sensor inputs, a chain of clusters, configured as an acyclic directed graph, will be involved. Typically, the chain will consist of one or more of each of the three types of clusters.

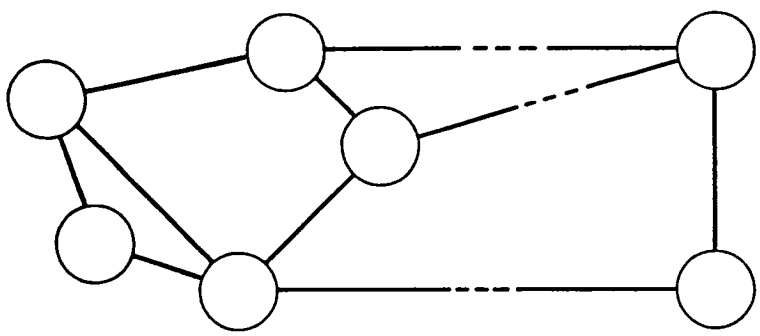


Figure 1. NETS is an Incomplete Interconnection of Clusters

Each cluster is a simplex (nonredundant) processor or a redundant processor. A redundant processor will most likely be configured as in SIFT, i.e., a complete interconnection among a set of processors. As discussed below, a cluster will require no more than 5 processors to meet the overall reliability requirements typical of an advanced aircraft.

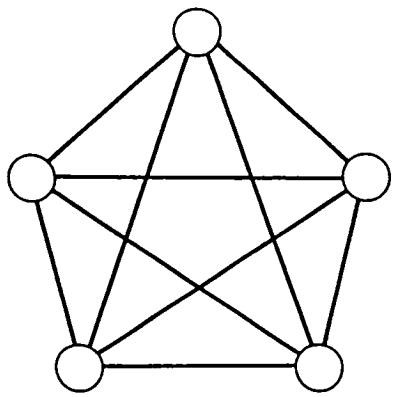


Figure 2. Within a NETS Cluster the Interconnection is Complete

Figure 3 depicts the structure of the interconnection between a pair of clusters (A and B), assuming that each of the clusters is a SIFT containing 5 processors (hereafter called 5-SIFTs). Note that the interconnection between A and B consists of only 5 links. In contrast, 10 processors configured as in SIFT (Each processor is connected to every other processor – the interconnection graph is *complete*), would require $\frac{10\times 9}{2} = 45$ links. An important question is: Can the reduced interconnection as provided by NETS meet the severe reliability requirements of an aircraft computer? As we show below, the answer is yes.

Assume that a task a executing on A is required to transmit data to a task b executing on B. It is assumed that a is computed on each of the working processors of A. Thus each of the A processors transmits its results of executing a to B using the link connecting it to B; a link, then, corresponds to an edge in the cluster interconnection graph. When all processors of B receive the results, they exchange the values received and perform a vote – as is standard for SIFT in processing input values. Thus link failures are masked, provided the number of good links exceeds the number of bad links remaining in the configuration. Note that a link fails when either of its associated processors fails. Furthermore, each cluster will become aware of its bad links and avoid using them – this is the usual adaptive voting technique.

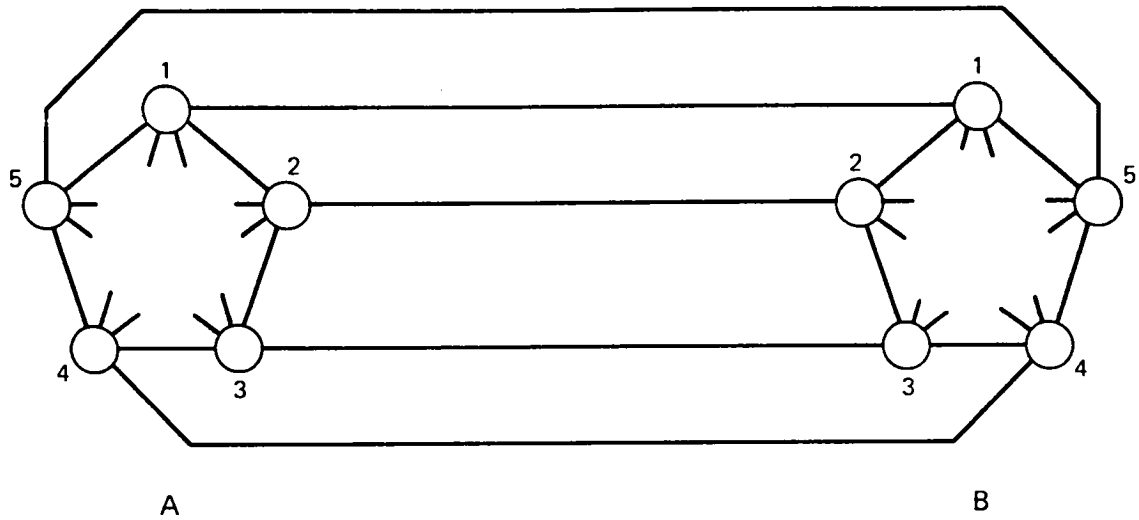
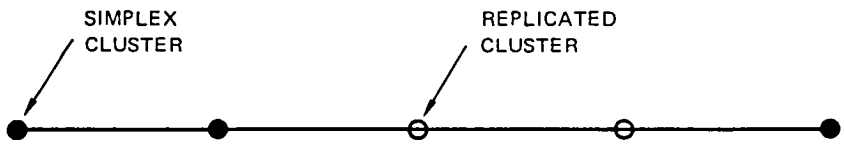


Figure 3. A 2-cluster NETS; 5 processors per cluster

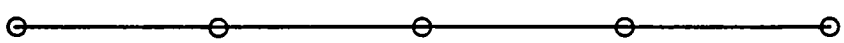
The discussion above is concerned with the case where cluster B contains a task that requires data from B's neighbor A. What if the destination of A's data is to be a third cluster C, but the route to C from A includes B? In this case, as above each of the working processors of B vote on the values received from A. The results of the vote are then transmitted to C using those links not known to be faulty. This *vote and forward* protocol is clearly appropriate for accommodating to both link and processor faults, but can be costly in terms of delay. Follow-on work will be concerned with analyzing the delay and with ways of minimizing the delay through assignment of tasks to clusters.

To guarantee masking in a chain, all of the chain's clusters must be replicated; it is anticipated that this kind of configuration would be employed for critical computations. Note that support for replicated sensors and actuators is inherent in this approach; each of the replicas would be associated with a processor. Further note that the replication must be preserved for those clusters whose role is only to forward data. Noncritical computations, on the other hand, need not involve a chain all of whose clusters are replicated. Of course, there is no harm in using

replicated clusters, other than a waste of computation power and an extra delay expended in carrying out the store-and-vote protocol. It is likely that this protocol can be dispensed with when the data to be forwarded originated with a noncritical task. Figure 4 summarizes the possibilities.



(a) Acceptable Path
End nodes – simplex
Intermediate nodes – simplex or replicated



(b) Acceptable Path
All nodes replicated

(c) Unacceptable Path
End nodes – replicated
Intermediate nodes – simplex or replicated

Figure 4. Communication Paths containing Simplex and Replicated Clusters

It is envisioned that each of replicated clusters will run the SIFT executive: local executive, vote, error report, global executive, etc. To manage the interfacing to the network, each cluster will also have a **network executive**, whose main functions will be:

- Forward messages destined for other clusters. Where necessary, the successful transmital of messages is to be acknowledged.
- Receive and process error reports from neighboring clusters, from which faulty links can be identified and avoided.
- Determine *optimal* paths to use in the communication of data between clusters, where optimal means shortest delay. It is conjectured that this determination can be carried *locally* in the sense that in deciding the shortest path to a cluster A', cluster A selects its neighbor B such that the length of the path from B to A' is shorter than than the path from any other of A's neighbor B' to A'.
- Participate in the initialization of neighboring clusters and in their recovery from massive transients that disable the entire cluster.

3.3 Reliability Assessment

In this section we consider the reliability achievable by NETS. It is shown that acceptable reliability – better than the requirement of 10^{-10}/hour – can be obtained for relatively large NETS systems by using 5-SIFT clusters with a fan-out not exceeding 3 from each cluster. We consider separately the following fault occurrences: (1) permanent faults – system failure due to exhaustion of spares, and (2) permanent faults – system failure due to buildup of faults before reconfiguration is completed, Future work will consider transient faults.

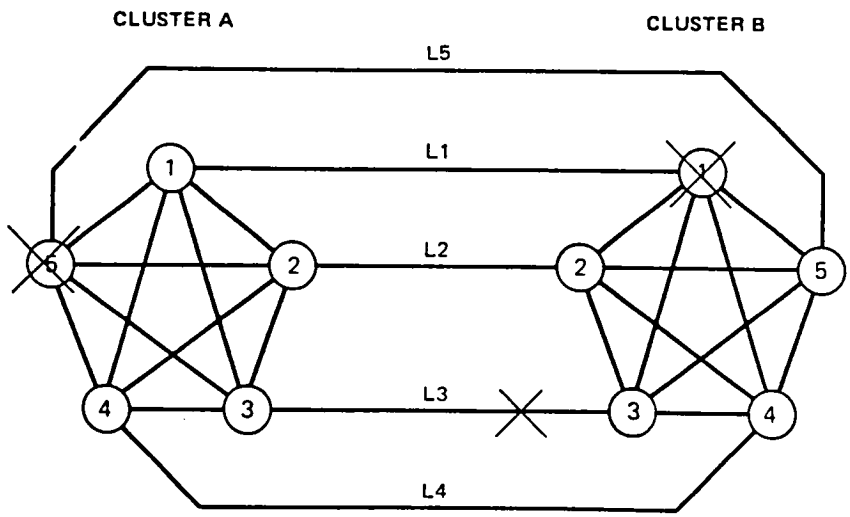
3.3.1 Permanent Faults – Exhaustion of Spares

We will carry out the analysis for this case by first indicating how the analysis is accomplished for simple cases: NETS containing 2 and 3 clusters. Then we will derive the general result. The concern here is with system failure due to link and cluster failures. It is assumed that the system fails if:

1. One or more clusters fail, or
2. Due to link failures, two clusters are unable to communicate with each other. This condition reduces to a cluster being unable to communicate with *any* of its neighboring clusters.

3.3.2 Two-Cluster System

First we will show that *all* patterns of three link failures can be tolerated. Next the number of patterns of 4 link failures leading to system failure will be enumerated. Then the probability of system failure will be approximated as the sum of the probabilities of link failures that cause system failure and cluster failures that cause system failure.

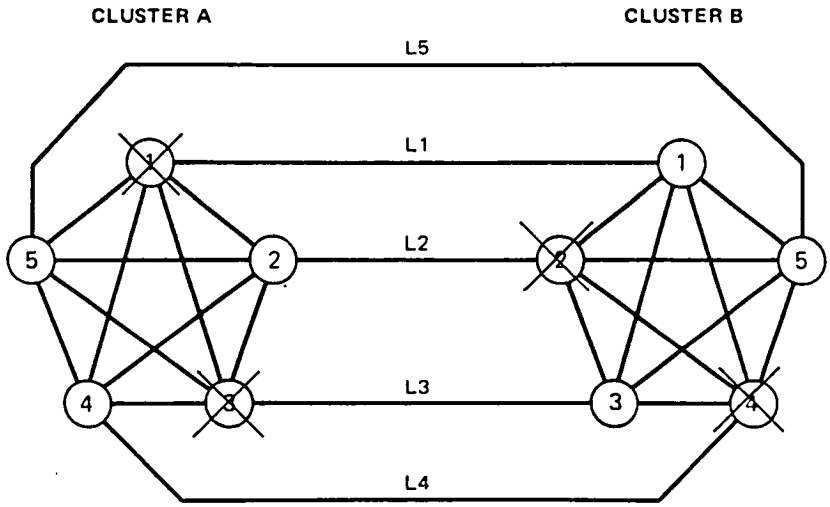


After handling failures in B1 and A5,
the system can tolerate failure of L3.

Figure 5. Failures tolerated in Two Cluster Nets.

We assume that link failures are detected immediately after their occurrence. Thus, referring to Figure 5, failures of processors B1 and A5, implying failures

of links L1 and L5, would be detected in turn and handled by the **network executive**. At this point, there are 3 working links L2, L3, and L4, allowing the system to tolerate another link failure – say L3; the results emerging from the two working links would outvote the result from the newly failing link.



Examples of untolerated patterns of 4 failures are:
A1,A3,B2,B4 A1,A2,A3,B4 and A1,A2,A3,A4.

Figure 6. Failures not tolerated in Two Cluster NETS.

Now, as depicted in Figure 6, let us consider patterns of 4 faults that lead to link failure causing system failure. We claim that faults in A1, A3, B2, B4 is such a fault pattern, as it implies failure of four links: L1, L2, L3, and L4. (Note that not all patterns of 4 faults lead to system failure; for example, faults in A1, A2, B1, and B2 would cause only 2 links – L1 and L2 – to fail.) Another pattern of 4 faults that leads to system failure is A1, A2, A3, and B4. It is easily shown that there are two classes of failures to be considered. For each of these classes communication between the clusters can no longer be guaranteed, although there are still adequate resources left in each of the clusters to allow voting to mask all internal cluster failures.

1. Failures of processors Ai, Aj, Bk, Bl, where i, j, k, l are all different (as illustrated in Fig. 6). Only one reliable link now exists between the two clusters (link 5 in the figure), in which case the voting of results transmitted between the two clusters no longer masks errors. The number of such patterns is $\binom{5}{2} \times \binom{3}{2} = 30$, where $\binom{n}{m}$ is the *combination* function – the number of combinations of n items taken m at a time.

2. Failures of processors Ai,Aj,Ak,Bl or Bi,Bj,Bk,Al, where i, j, k, l, are all different. Similar to the situation in (1), only 1 reliable link (link 5) remains for intercluster communication. The enumeration here yields $2 \times \binom{5}{1} \times \binom{4}{3} = 40$ failure patterns.

(Note that a failure of 4 processors all within a cluster is not considered here as it implies a failure of the cluster itself – see below) Summing (1) and (2) yields 70 failure patterns or a failure probability of approximately $70p^4$, where p is the probability of failure of an individual processor.

A cluster itself fails when the number of operational processors within a cluster is inadequate to permit error masking through voting. Assuming, as above, that faults occur at a low enough rate to permit the logical removal of a faulty processor before the occurrence of a subsequent fault, failure of a cluster occurs when 4 faults are occur. The probability of 4 faults within a cluster is approximately $\binom{5}{4} \times p^4$ or $10p^4$ when both A and B clusters are considered. Thus the probability of system failure due to link failure and the probability of system failure due to cluster failure are approximately the same.

3.3.3 Three Cluster System

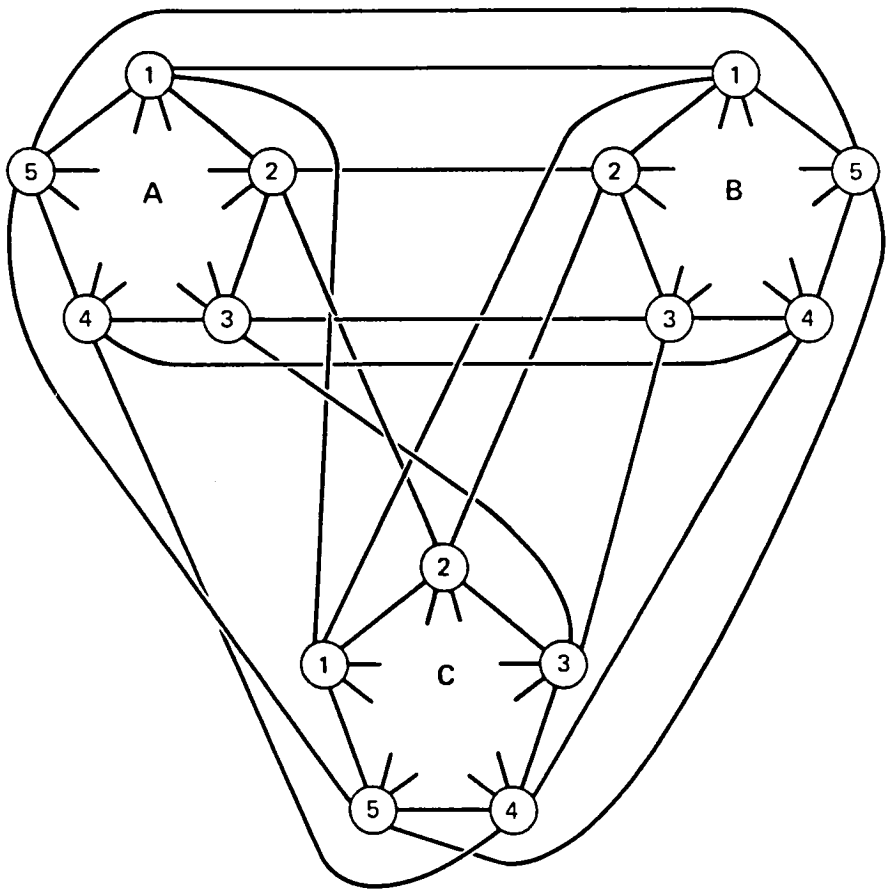


Figure 7. A Three Cluster NETS

Figure 7 depicts a 3-cluster NETS system; the fan-out from each cluster is two. As in the 2-cluster system, all patterns of three link failures are tolerated, but here, with the use of routing of broadcasts via intermediate cluster, all patterns of four link failures are also tolerated. Figure 8 illustrates the protocol for communication in the presence of the following faults: A1, B2, B3, and B4. This fault pattern prevents A and B from communicating with each other directly. However, they can communicate through C. If A is to communicate with B, the four working processors of A will send data to C where the five working processors

will exchange the values received from A and, after voting, come to an agreement on the values sent. C, using the two good links (1 and 5) that connect it with B will transmit the data to B; this means that of C's 5 working processors, only C1 and C5 will participate in the communication with B. It can be shown, then, that no pattern of 4 processor failures will cause link failure.

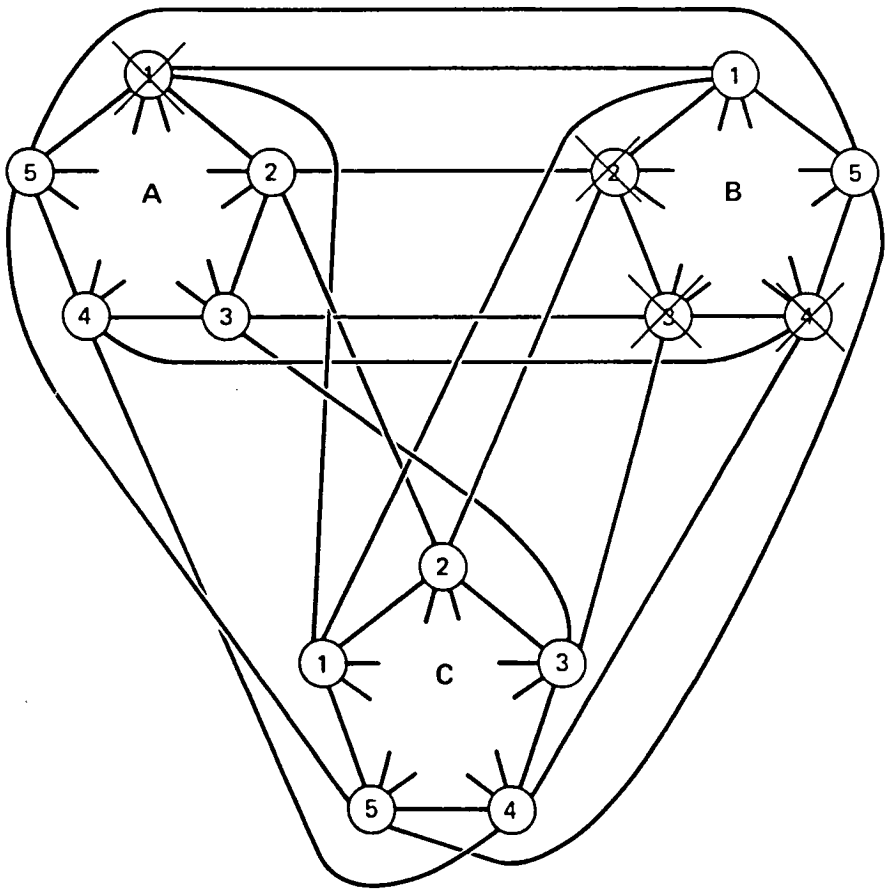


Figure 8. A pattern of four faults that does not cause Link Failure

However, there are patterns of five failures that are not tolerated. The typical pattern of such an untolerated fault pattern is: Ai, Aj, Ak, Bl or Bm, Cl or Cm, where i, j, k, l, m are all different, as shown in Figure 9. The number of such patterns is $3 \times \binom{5}{3} \times \binom{2}{1} \times \binom{2}{1}$ yielding a failure probability due to link failures of approximately $120p^5$.

Note that the probability of a cluster failure (again, due to exhaustion of spares) is $15p^4$ which, except for unreasonably high values of p, is larger than the probability of link failure. Thus, for the case of a 3 cluster system, the assumed interconnection is adequate with respect to achieving reliability in the presence of failures that result in exhaustion of spares.

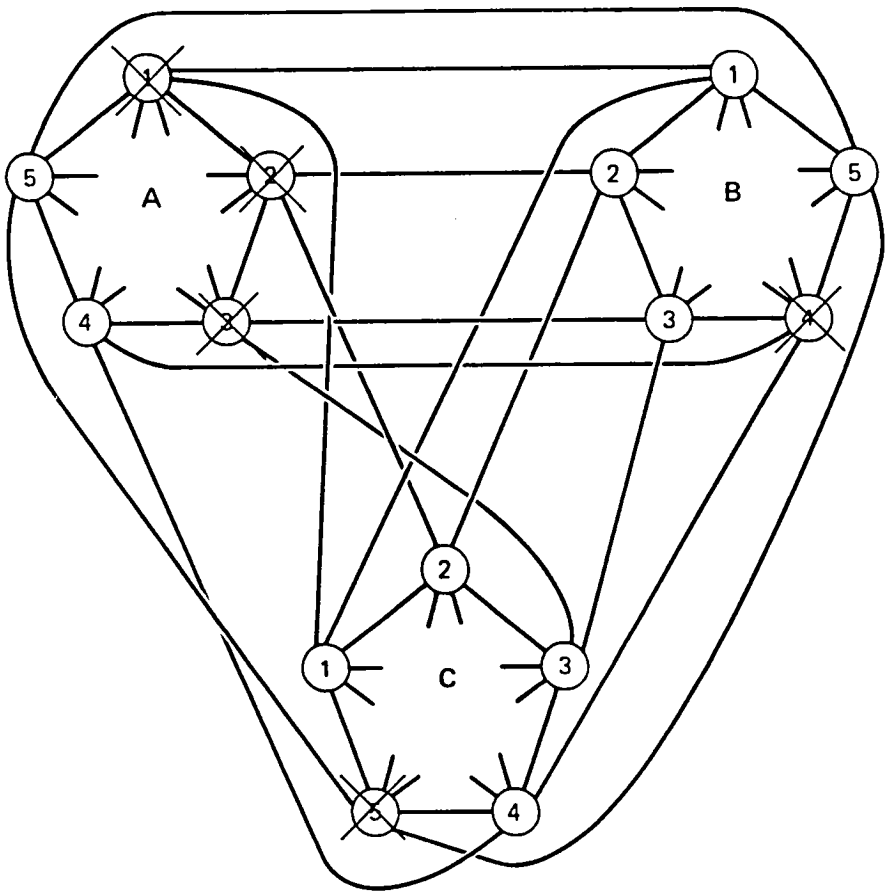


Figure 9. A pattern of five failures that is not tolerated.

3.3.4 General Result for N-Cluster Systems with Fanouts of 2 or 3

Consider a fan-out of 2 from each cluster. System failure will occur when one (or more) clusters is unable to communicate with any neighbor, thus isolating it from the rest of the system. Enumerating the faults that cause this situation, we find that the probability of system failure due to link faults is given by $N \times 40 \times p^5$. (This result is easily derived as a generalization of the enumeration carried out for the the 3 cluster NETS.) The probability of system failure due to cluster failure is $N \times 5 \times p^4$; thus, for this general case, the fan-out of 2 is adequate for achieving reliability.

If the intercluster fan-out is increased to 3, the probability of system failure due to link failure is decreased to $N \times \binom{5}{3}\binom{2}{1}^3 \times p^6 = N \times 80 \times p^6$.

3.3.5 Permanent Faults – Fault Buildup Prior to Reconfiguration

The previous subsection considered the case where faults occur at a low rate, thus permitting the system to reconfigure itself after each fault occurrence. However, a complete reliability analysis must consider the case where faults are not handled immediately, allowing faulty processors to remain in the configuration. We have not completed the analysis here, but our preliminary results are as follows.

A cluster itself will fail if 3 processors fail before reconfiguration can be carried out. The probability of at least one such cluster so failing is $N \times 10 \times p^3$. For link failure, the typical pattern containing 4 faults causing failure before reconfiguration can be completed, is Ai, Aj, (Bk, Bl, or Bm), (Ck, Cl, or Cm). Thus the probability of system failure due to link failure is $N \times 90 \times p^4$, again significantly lower than the probability due to cluster failure.

3.4 Further Work

Additional work is recommended on the NETS concept to enable an assessment of its suitability for advanced aircraft. Particular issues to be considered are the following:

1. Transient Fault Analysis: A transient fault causes a processor to temporarily deliver erroneous results. After a period of time, the processor will return to a state where it will deliver correct results. The usual technique for dealing with transient faults is permit the processor suffering the fault to deliver the erroneous values for a certain period of time t, during which voting will mask the error. If the processor does not return to an error-free state within t, it is considered to have suffered a permanent fault. If t is set at too low a value, then long-duration transient faults will be considered as permanent faults, and good processors will be removed from the system. On the other hand, if t is set too high, transient (and permanent) faults can build up, causing the system to fail by having the voting margin exceeded. To determine the vulnerability of NETS to transient faults, it will be necessary to weigh the probability of exceeding the voting margin against the probability of running out of spares.

2. Optimal graph structures for nets interconnection network: The goal here is to minimize the delay associated with a computation, where the primary controllable contribution to delay is in intercluster communication. One possible objective function is to minimize the maximum delay by having an appropriate structure for the network. There is a class of graphs, called (n,d,k) graphs, the property of which is that for a graph of n nodes and a fanout of k, the maximum distance between any pair of nodes is d. Here distance is the number of hops required to go between *any* pair of nodes, where the shortest path is selected. One objective is to maximize n for given values of d and k. Kautz [5] has identified the graphs for different values of n,d, and k. It remains to determine if the measure of minimum d is appropriate for optimum communication in NETS. Another issue is the assignment of tasks

to nodes to minimize the delay. One additional issue to study is to define what we call (n,d,k,f) graphs, graphs in which the distance between any pair of nodes is no more than d under the assumption that up to f link failures have occurred.

3. A Distributed Algorithm for Computing Optimal Communication Paths: It will be necessary to associate paths in the interconnection graph with each pair of clusters that communicate with each other. The computation of such optimal paths can be done "offline" for the initial configuration. As link failures occur, certain paths might become closed off, in which case it will be necessary to determine new paths. It is conjectured that this determination of optimal paths can be accomplished in a local manner as follows: If cluster A can no longer use B in communicating with C, it chooses to communicate with C using that neighboring cluster D such that the distance between D and C is smaller than the distance between any other neighbor and C. It remains to verify this conjecture. Also, we must consider the impact of a second failure while the new shortest path is being computed.

4. Synchronization in NETS: It will be necessary for the clusters to be synchronized with each other, although more skew might be acceptable among clusters than among the processors within a cluster. Also the skew might not have to be uniform over the network: clusters distant from each other might be able to have more skew than clusters close to each other. It has been shown that the fan-out must be at least 2 to allow the clusters to synchronize themselves.

5. Recovery of a Cluster After a Massive Transient: We are defining a *massive transient fault* as a condition where one or more clusters suffer faults in their processors to the point where their voting margins are insufficient to mask the processor failures. When a cluster has suffered a massive transient fault, it is likely to be unable to recover without assistance. In NETS, the assistance can come from neighboring clusters. The tentative approach we have developed for effecting recovery is as follows. The steps to be followed are: *checkpointing, failure detection, agreement on failure* and *rebooting.* In checkpointing, each cluster will keep a current record of the failure status

of the processors in neighboring clusters (distance 1 away); it is noted that this record is essential to determining which links are still working. Failure detection involves noting that a cluster is misbehaving to the point where it fails to deliver outputs or fewer than half of the processors in the cluster are producing identical values; to distinguish between a massive transient and a buildup of permanent failures, a sudden occurrence of errors is likely to indicate a massive transient. Agreement on a cluster c having suffered a massive transient is achieved by all neighbors of c carrying out an interactive consistency procedure on the state of c. Rebooting is achieved by all working neighbors of c restarting each of c's processors which were working before the massive transient and indicating to each of them those processors that are working and the links that are operative. It is conjectured that as long as the interconnection graph remains connected (there is a path between each pair of clusters), recovery can be obtained with the above mentioned procedure.

Chapter 4

Asynchronous Voting

The need for reliable computation has induced many designs for fault tolerant computer systems based on the replication of the processors and appropriate error detection and masking algorithms. Typical of such systems are SIFT and FTMP, which use majority voting for error masking, and Stratus, which uses a dual-dual structure for error masking. It is clear that these approaches, coupled with the steadily improving reliability of components, now allow the construction of very reliable systems.

All fault tolerant systems depend on some form of error masking algorithm, coupled with error detection to allow the repair of faults. Some such systems depend on backward error correction, in which a result is computed, the acceptability of that result is checked, and in the event of error the computation of the result is repeated. Typical of such systems are classical Checkpoint-Restart systems and Recovery Blocks. Backward error correcting algorithms necessarily incur a significant overhead for repeating the computation when an error is detected, and also involve an acceptance test on the results, a test that is usually system and application specific. We do not consider backward error correcting systems in this paper but rather we examine Forward Error Correcting systems, in which

the results are computed in a redundant form that allows error masking without repeating any computation.

Two forward error correcting algorithms are currently used for masking processor errors in reliable systems, majority voting and dual-dual. The majority voting approach can mask errors caused by one faulty channel out of three, while a dual-dual approach masks one faulty channel out of four. Both approaches have the advantage that they are completely application independent. However majority voting and dual-dual both depend for their operation on exact match comparison between results of computations. Thus, for successful masking of errors, it is essential that the fault free channels should generate identical results. Both algorithms guarantee, with only a single faulty channel and with fault free channels producing identical results, that fault free channels remain error free and continue to generate identical results.

Two questions arise from this. The first concerns whether there are any single point faults that could cause fault free channels to generate different results, thus invalidating the presumptions of both majority voting and dual-dual. We describe below a class of such faults and give algorithms for precluding them. The second question relates to the possible increase in the risk of common mode faults resulting from the need for all channels to perform exactly the same computation on identical data at approximately the same time. We show below that error masking algorithms can be devised that allow each channel to perform a different computation on different data at different times.

4.1 Loss of Consistency

Figure 10 shows a majority voted three channel system, with one faulty and two working channels. The successive levels of the diagram might represent distinct units within the channel, but equally they can represent successive iterations of a computation performed by the same units. It is clear that, provided that the two working channels generate identical results initially, each voting operation

will receive as inputs two identical values and one erroneous value. The voters in the two working channels will therefore both produce the same value for the majority. Thus the working channels continue to generate identical results, and consistency between working channels is maintained. However, if at any time the three channels generate different results, the voters can find no majority and the system fails.

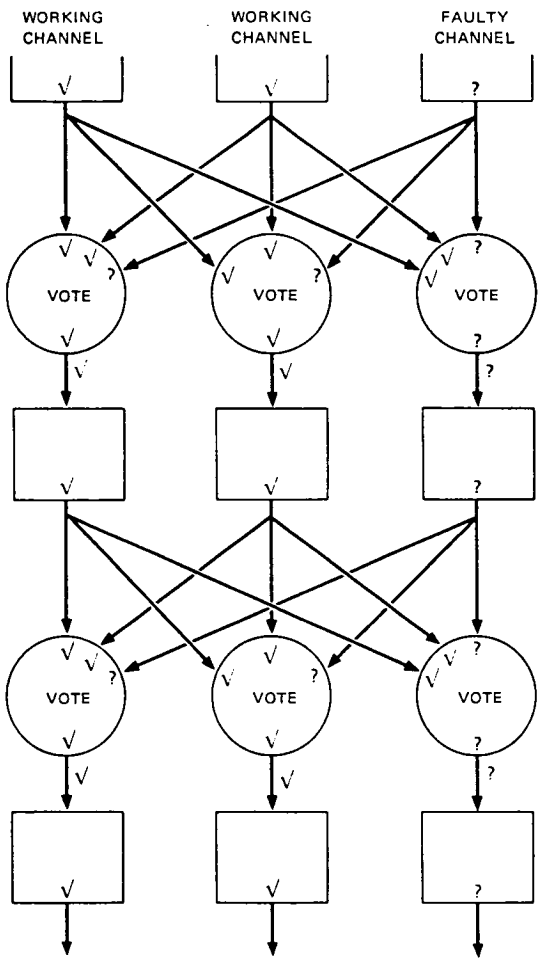


Figure 10. A Three Channel Majority Voted System

Consider Figure 11, which shows a system of three working channels and an input to that system from a single faulty source. The nature of the fault is that the source distributes different values to each of the three channels (the values A, B, and C). Even on a broadcast bus, such faults can result from marginal timing faults or from a marginal transmitter at the source and receivers with slightly different, but within specification, characteristics. More complex communication mechanisms, particularly where software is involved, permit many more such faults. The figure shows that, if the faulty source distributes different values to each channel, the three channels generate different results, the voters can find no majority, and the system fails.

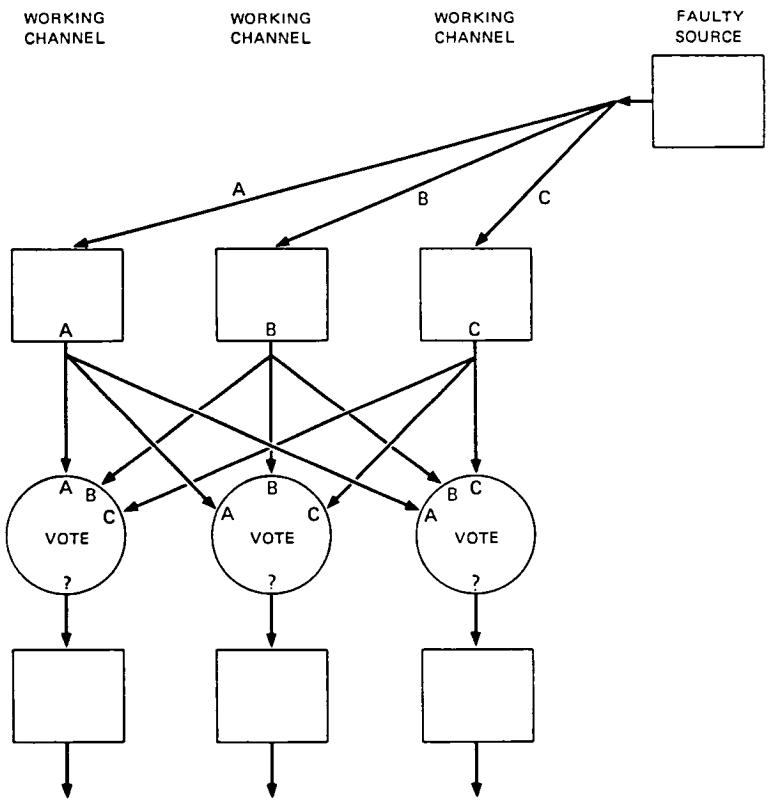


Figure 11. Distribution of Information from a Single Faulty Source to a Three Channel System

Figure 12 shows a three channel system with two working and one faulty channels. Here information present in just one of the channels is to be distributed to all three channels and be used in a replicated calculation. The faulty source distributes different values to the two working channels, and compounds the problem by repeating the same erroneous values (suitably transformed if necessary) in the next, voted, stage of the system. Note that not only do the two working channels continue to receive inconsistent values, even after voting, but also each of the two working channels can be mislead into believing that it is the other working channel that is faulty.

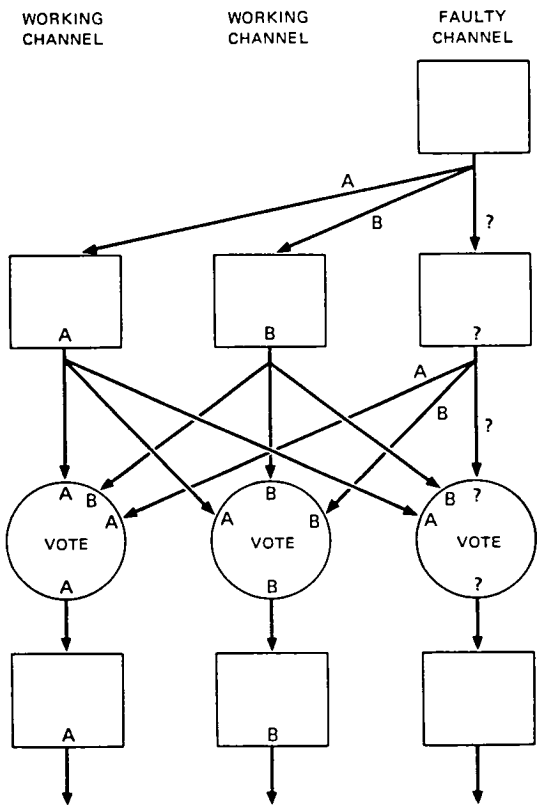


Figure 12. Distribution of Information from a Single Channel to Three Channels

The existence of this problem was discovered during the design of SIFT, a reliable aircraft control system, and is discussed in [4], where it is shown that no solution is possible in a purely three channel system. An algorithm, called the interactive consistency algorithm, is given for a four channel system containing a single faulty channel, and extended to the masking of N faults in a 3N+1 channel system.

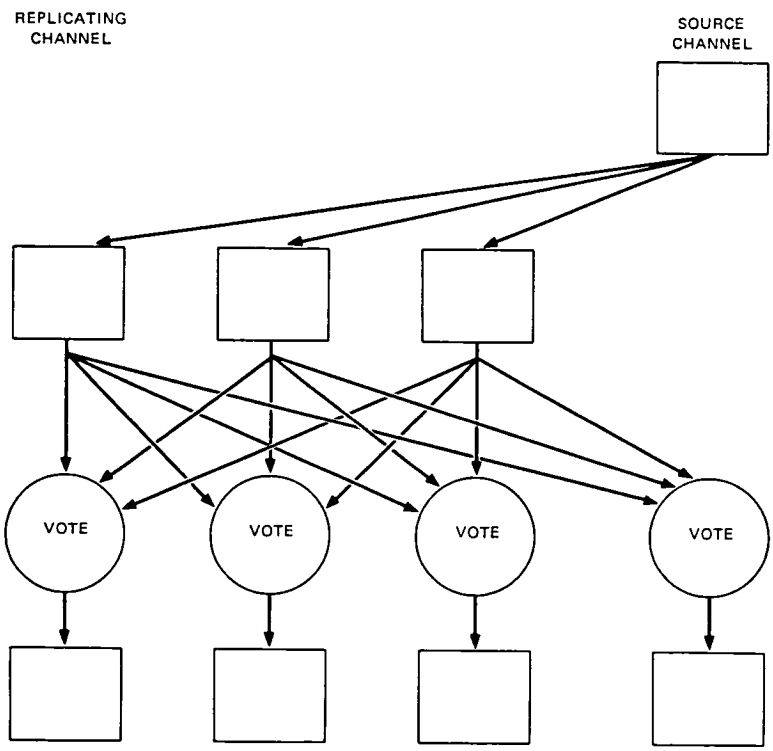


Figure 13. The Interactive Consistency Algorithm

The basic interactive consistency algorithm is given in Figure 13. One of the four channels is the single point source of the information, and the three other channels are used to replicate that information. Once the information is replicated, any or all of the channels can vote the replicated information with confidence that all voters in working channels will produce the same majority value, or alternatively all working voters will find no majority and will return a default value. For this algorithm to be effective against all faults, the channel that is the source of the information must be distinct from the three channels that perform the replication.

Consider the possibility that the source channel is faulty. It may then distribute different values to the other channels. The three replicating channels must all be working, and thus every working voter must get the same set of inputs. If at least two of the replicating channels have the same value, every working voter will find that value as its majority, while if all three replicating channels have different values, every working voter will return the default value. (If the source is faulty, the interactive consistency algorithm cannot of course guarantee a correct value from that source, but only a value that is consistent across all working channels.)

Consider the possibility that one of the three replicating channels is faulty. Now the source is necessarily working and will distribute the same correct value to each of the two working replicators, which will replicate it. Thus each working voter obtains at least two correct inputs and is able to produce the correct value as its result.

In SIFT, four circumstances were found in which a value from a single source had to be distributed to three replicated channels, namely:

- input from a sensor,
- error reports from a voter,
- interfaces between unreplicated and replicated tasks,
- synchronization of processor clocks.

The first three of these require the use of the interactive consistency algorithm to protect the system against malicious faults. The fourth is of special interest in that exact agreement is not necessary for clock synchronization, and thus slightly simpler algorithms guaranteeing approximate agreement suffice.

4.2 Maintenance of Approximate Consistency

In SIFT, as in many other fault tolerant systems, each processor has its own clock and operation of the system depends on these clocks remaining synchronized (to within 50ms in SIFT). Many prior systems used three channels, three clocks, and a clock synchronization algorithm based on each clock synchronizing itself periodically to the median clock of the three. It is instructive to consider why this "obviously sound" approach is invalid.

Figure 14 shows a system with two working clocks (A and B) and a faulty clock (C). We may assume that clock A runs slightly faster than clock B. Clock C presents to clock A an erroneous clock value indicating that clock C is running faster even than clock A, causing clock A to assume that it is the median clock. Thus clock A makes no correction to its value. Similarly, clock C presents to clock B a value indicating that it is behind even clock B, causing clock B to assume that it is the median clock and make no correction to its clock value. By this strategy, the faulty clock C can induce clocks A and B to operate without correcting their clock values as they gradually drift apart until the system fails. Single point component faults that could cause this "malicious" behavior have been found even in purely analog clock systems.

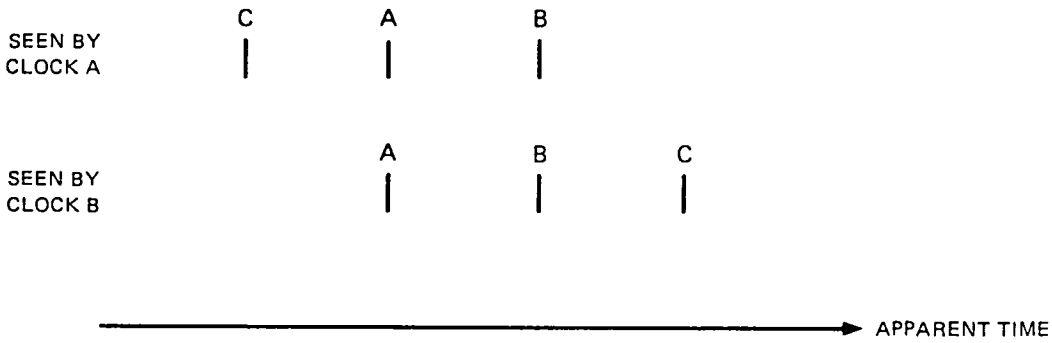


Figure 14. A Failure Mode of the Median Clock Synchronization Algorithm

It is tempting to attempt minor corrections to the three channel clock synchronization algorithms, aimed at preventing this behavior. As yet we have no rigorous mathematical proof that no three channel algorithm can exist, but we believe that the approximate agreement needed for clock synchronization requires the same number of channels as the exact agreement discussed above.

In SIFT, a four channel clock synchronization algorithm is used in which each clock is periodically resynchronized to the mean of the four clocks. To protect against wildly erroneous clock values, the algorithm imposes a bound within which a clock value must lie to be included in the averaging calculation. For n processors of which at most m are faulty, with R as the resynchronization interval and S as the time taken for resynchronization, and if ϵ is the maximum clock reading error and ρ the maximum rate of clock drift, it can be shown that the maximum skew between working clocks will not exceed

$$\frac{n}{(n-3m)}(2\epsilon + \rho(R + \frac{2(n-m)S}{n})).$$

A similar problem has been examined by L. Webster [7, 8] in closed loop control systems. He found that use of a median voting algorithm in a three channel system favors the median channel, effectively disconnecting the two other channels from the closed loop. Without cross coupling between the integrators

of the three channels, this results in uncontrolled accumulation of error terms in the integrators of two of the channels, rendering them useless for error masking. With cross coupling, the integrators are vulnerable to precisely the same problem as the clocks above.

The possibility of failure to maintain approximate consistency appears to exist in any three channel system containing embedded integrators.

4.3 Asynchronous Multichannel Systems

Existing fault tolerant multichannel systems using forward error correction, whether majority voted or dual-dual, depend on an exact equality between the result values of the various channels. To ensure this exact equality of their outputs, the various channels must all perform exactly the same calculation on exactly the same input values at approximately the same time. This exposes such systems to an unquantifiable risk of correlated faults generating errors simultaneously in several channels. Such correlated faults might result from some external influence, such as lightning or cosmic rays, or from accumulation of latent faults not within the coverage of the diagnostics, or from design faults in the hardware logic or the software.

A much higher degree of confidence in the resilience of the system to correlated faults would result from a system design in which each channel performs its calculation at different times, on different input values, and obtains different outputs. It is even possible to consider the use of different algorithms in each of the channels. Unfortunately, as exhibited above, without an exact match between channels, standard voting techniques are vulnerable to faults that cause loss of consistency between channels and thus system failure. We seek here to provide alternative algorithms that permit differences between channels without risk of loss of consistency.

The first thoughts on an approach to such asynchronous error masking envisage a system of four channels. Each channel operates at the required iteration

rate but completely unsyschronized with the other channels, thus minimizing interaction between channels. Each result produced would carry a timestamp. A processor, when voting such a result, would have access to the four most recent values, one from each channel, together with their timestamps. From these it would be possible to extrapolate to a most probable current value, as shown in Figure 15.

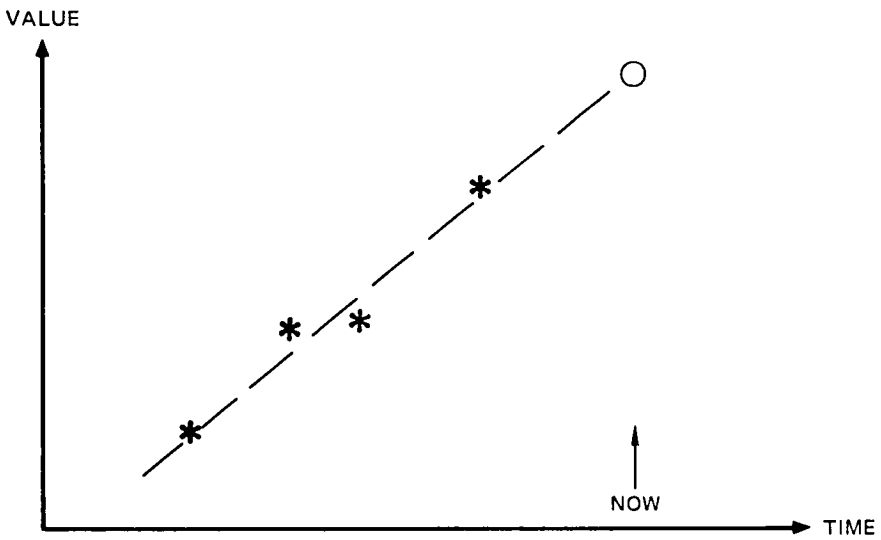


Figure 15. Extrapolation from Past Values to a Most Probable Current Value

More formally, if $R_{i,p}$ is the i'th broadcast result from processor p, containing a value $v_{i,p}$ and a timestamp $t_{i,p}$, and if the most recent result so far received from processor p is n_p, the algorithm can be expressed as:

$$\text{consensus value} = F(v_{(n_a,a)}, t_{(n_a,a)}, v_{(n_b,b)}, t_{(n_b,b)}, v_{(n_c,c)}, t_{(n_c,c)}, v_{(n_d,d)}, t_{(n_d,d)})$$

where F is some function to be determined, and a, b, c, d are the four processors.

Unfortunately, it is easy to show that the timestamps do not assist in the maintenance of consistency in the absence of any constraints on the times at which results are calculated. If greater weight is given to more recent values, those values may be erroneous values increasing the vulnerability of the system. In particular,

consider the case in which three good values are reported approximately simultaneously and subsequently an erroneous value is reported. Any preference given to recent values can only render the consensus less reliable than that obtained by ignoring the timestamps.

Consideration can also be given to the clock synchronization algorithm described above. Here, if processor a is considering the values generated by processors b, c, d, with current values v_a, v_b, v_c and v_d,

$$\text{For } i \text{ in } b,c,d : v'_i = \begin{array}{l}\text{if } v_i > v_a + \delta \quad \vee v_i < v_a - \delta \\ \text{then } v_a \\ \text{else } v_i\end{array}$$

$$\text{and then:} \quad \text{consistent result} = \frac{v_a + v'_b + v'_c + v'_d}{4}$$

That algorithm does indeed maintain consistency between channels, but the rate of convergence is very weak and the drift and error signals that can be introduced by undetected faulty clocks are much larger than the permitted drift and jitter of working clocks. In the clock synchronization application this is not critical for the individual clocks have performance characteristics much better than those required for typical system applications. For a control system application however, the errors introduced by a faulty channel can easily overwhelm the control action of the system, and thus such an algorithm is clearly unacceptable.

A possible alternative approach requires that the four channels compute their results at uniform phases within the iteration interval, one channel generating a value at the start of the interval, a second channel generating its result a quarter of the interval later, etc., as shown in Figure 16. This additional information allows the algorithm an improved ability to compute a most probable current value and to reject erroneous values. The uniform spacing at which results are generated through the interval greatly simplifies calculations compared with a

system in which such spacings are arbitrary, and thus assists in reducing the voting calculation overhead.

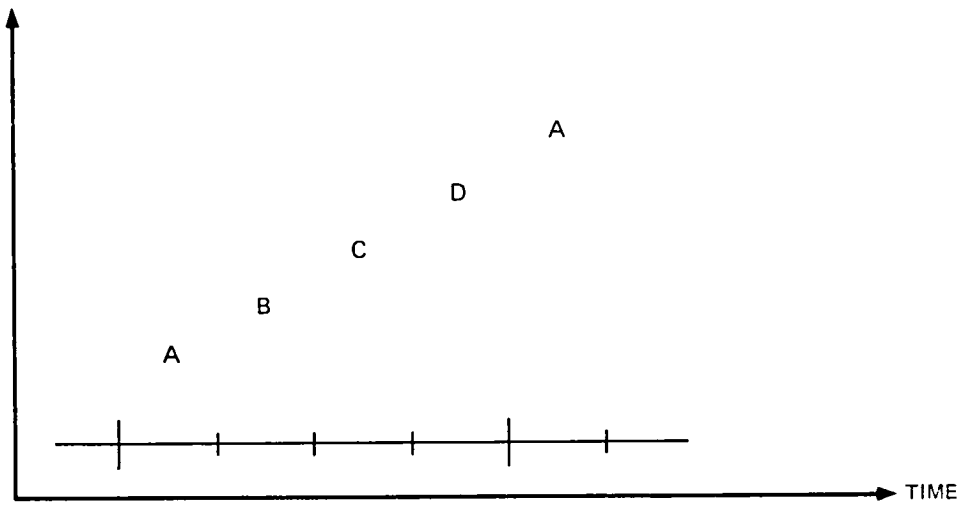


Figure 16. Calculation of Results at Uniform Phases within an Interval

An initial evaluation of such a system was made, using the arithmetic mean of the four values for the most probable current value, as in the clock synchronization algorithm. Each channel uses fixed limits for the acceptable deviation of the values computed by other channels from its own most recent value, but those limits can differ for each of the other channels. Thus if δ is an appropriate acceptable deviation for the channel whose result was computed one quarter of an iteration later, then 1.3δ is an appropriate limit for the channel computing half an iteration later and 1.2δ for the channel computing three quarters of an iteration later. These slightly larger values are permissible because the algorithm gives greater weight to more recent values, though this must be balanced against the effect of an earlier erroneous value augmenting its disturbance by influencing the intermediate values.

Here, if processor a is considering the values generated by processors b, c, d, with current values v_a, v_b, v_c and v_d,

$$v'_b = \text{if } v_b > v_a + \delta \quad \vee v_b < v_a - \delta$$
$$\text{then } v_a$$
$$\text{else } v_b$$

$$v'_c = \text{if } v_c > v_a + 1.3\delta \quad \vee v_c < v_a - 1.3\delta$$
$$\text{then } v_a$$
$$\text{else } v_c$$

$$v'_d = \text{if } v_d > v_a + 1.2\delta \quad \vee v_d < v_a - 1.2\delta$$
$$\text{then } v_a$$
$$\text{else } v_d$$

$$\text{and then:} \quad \text{consistent result} = \frac{v_a + v'_b + v'_c + v'_d}{4}$$

Unfortunately, while this algorithm appears to be better than the basic clock synchronization algorithm, it is only slightly so and the drift and error signals introduceable by a fault are still at least comparable to the maximum permissible control action of the system. Thus the algorithm is still unacceptable.

We can refine the algorithm by giving different weights to each of the values, for instance:

$$\text{consistent result} = \frac{v_a + 2v'_b + 3v'_c + 4v'_d}{10}$$

but the effect is marginal and still far from providing acceptable margins for control purposes.

Error masking algorithms such as these act as filters and, like all filters, necessarily introduce delay into the control loop. The algorithms above introduce a

delay of about 2/3 of an iteration. To maintain the same margins of loop stability, the introduction of such a delay would require an increase in the iteration rate of about 33%.

A number of possible improvements to the algorithm are under consideration. We are currently working on algorithms that make better use of the relative timing of results, both by giving greater weight to more recent results in estimating the most probable current value, and also by considering the values generated by other channels when determining the acceptability of a result. A further possibility is the use of a five channel system fully capable of rejecting the most malicious faults which degrades on the first reconfiguration to a four channel system capable of rejecting all faults except those malicious faults in which different information is delivered to different destinations by the broadcast mechanisms. Since the probability of a second fault during a mission is low, and the probability of a malicious fault is also low, such a system might be judged to be adequately reliable.

References

[1] J. Wensley, et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE,* 60(10):1240-1254, October 1978.

[2] A. Hopkins, T.B. Smith, J. Lala, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft", *Proceedings of the IEEE,* 66(10): 1221-1239, Oct 1978.

[3] L. Lamport, P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults", in preparation, SRI International, Feb 1982.

[4] M. Pease, R. Shostak, L. Lamport, "Reaching Agreement in the Presence of Faults", *JACM,* 27(2):228-234, April 1980.

[5] W. H. Kautz et al., "Cellular Logic Networks and Machines", *IEEE Trans on Computers,* C-17(5):443-451, May 1968.

[6] Selim G. Akl, "Digital Signatures: Atutorial Survey", *Computer,* 16(2):15-26, February 1983.

[7] Dun, W. &Meyer, G., "A Fault Tolerant Distributed Micro-computer Structure for Aircraft Control Systems", *AIAA Guidance and Control Conference,* Palo Alto, Aug 1978.

[8] Dun, W. &Meyer, G., "Design and Analysis of a Fault Tolerant Distributed Microcomputer Control System", *IEEE Conference on Decision and Control,* San Diego, Jan 1979.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
172226		

4. Title and Subtitle	5. Report Date
Fault-Tolerant Architecture for Integrated Aircraft Electronics Systems	August 1983
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
K. Levitt, P. M. Melliar-Smith, R. L. Schwartz	SRI Project 4616
	10. Work Unit No.
	Task 1

9. Performing Organization Name and Address	11. Contract or Grant No.
SRI International 333 Ravenswood Avenue Menlo Park, CA 94025	NAS1-17067
	13. Type of Report and Period Covered
	Task Final Report

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
NASA Langley Research Center Hampton, VA 23665	

15. Supplementary Notes

16. Abstract

This report describes work into possible architectures for future flight control computer systems. The topics covered include:

*Ada for Fault-Tolerant Systems,

*The NETS Network Error-Tolerant System architecture,

*Voting in asynchronous systems.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Reliable system, fault tolerance, NETS, Ada, asynchronous systems	

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	56	

N-305

For sale by the National Technical Information Service, Springfield, Virginia 22161